En el capítulo anterior introdujimos muchos conceptos nuevos sobre las señales analógicas y la técnica de modulación por ancho de pulso. En este capítulo, apoyándonos en lo aprendido en el capítulo anterior, también generaremos señales PWM, pero las aplicaremos a un contexto totalmente diferente: la creación de sonidos. Ojo, va a ser un capítulo muy ruidoso.

El material para este capítulo se reduce a un único dispositivo externo:

* Un altavoz o un piezo pasivo.

8.1 Proyecto “Música arcade”

La música de un videojuego, igual que la banda sonora de una película, desempeña un papel que va más allá del mero acompañamiento de la acción. La música potencia las emociones que provoca el juego, crea ambientes, completa la experiencia creada por las imágenes y la interactividad... Y, sin embargo, en sus inicios los videojuegos eran silenciosos. Las limitaciones de los dispositivos en los años 50 y 60 del siglo XX hacían muy complicado compaginar imágenes y sonido, por lo que la música se restringía a la introducción del juego y a la pantalla de *game over*. A finales del año 1972 se produce el primer hito en la historia de la música en los videojuegos: se comercializa la máquina arcade Pong, que incluye efectos sonoros que se ajustan a la acción del juego. Space invaders (1972), Pac-Man (1980), Super Mario Bros. (1985) o Tetris (1989) nos remiten a melodías inolvidables en la historia de los videojuegos.

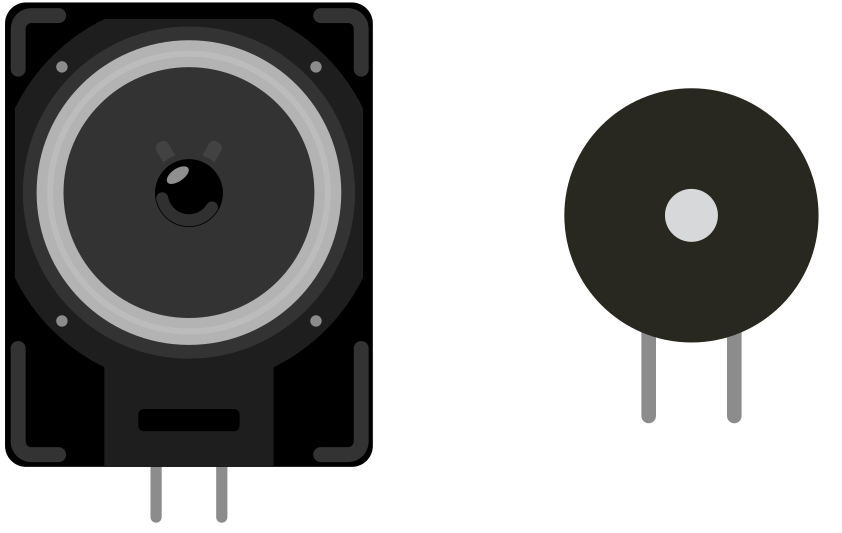


Aunque para la música y los efectos sonoros se usaba un chip diseñado específicamente para producir sonido, el timbre característico del sonido de los primeros videojuegos se puede recrear fácilmente con un altavoz y un microcontrolador. Y eso es precisamente lo que vamos a hacer en este proyecto: reproduciremos las melodías de varios juegos de arcade míticos.

8.2 Sonidos electrónicos, piezos y altavoces

La archifamosa melodía del Tetris, el pitido del lavavajillas que indica que el lavado ya ha finalizado, la alarma del despertador… Todos estos sonidos tienen algo en común: son sonidos electrónicos. El motivo de llamarlos así es que han sido generados de manera electrónica. En lugar de usar los nudillos para golpear una puerta o deslizar el arco por las cuerdas de un violín, los sonidos electrónicos se crean usando circuitos alimentados por una corriente.

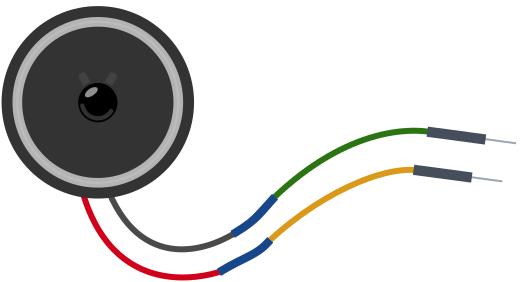
Los sonidos electrónicos son emitidos por zumbadores piezoeléctricos (también llamados piezos o zumbadores) o por altavoces. Para generar un sonido electrónico con un microcontrolador, en el pin donde está conectado el altavoz se crea una señal de la frecuencia deseada. Esta señal hace vibrar la membrana del altavoz con la frecuencia programada y la vibración de la membrana del altavoz se transmite por el aire en forma de onda sonora que llega a nuestros oídos.



Altavoz (izquierda) y zumbador piezoeléctrico (derecha).

A efectos prácticos, los piezos y los altavoces funcionan de la misma manera, pero en el desarrollo de este capítulo utilizaremos un altavoz porque ofrece mejor calidad de sonido. Si usamos un piezo tenemos que asegurarnos de que sea pasivo; los piezos activos ya vienen programados para emitir sonido de una determinada frecuencia, así que no nos valen.

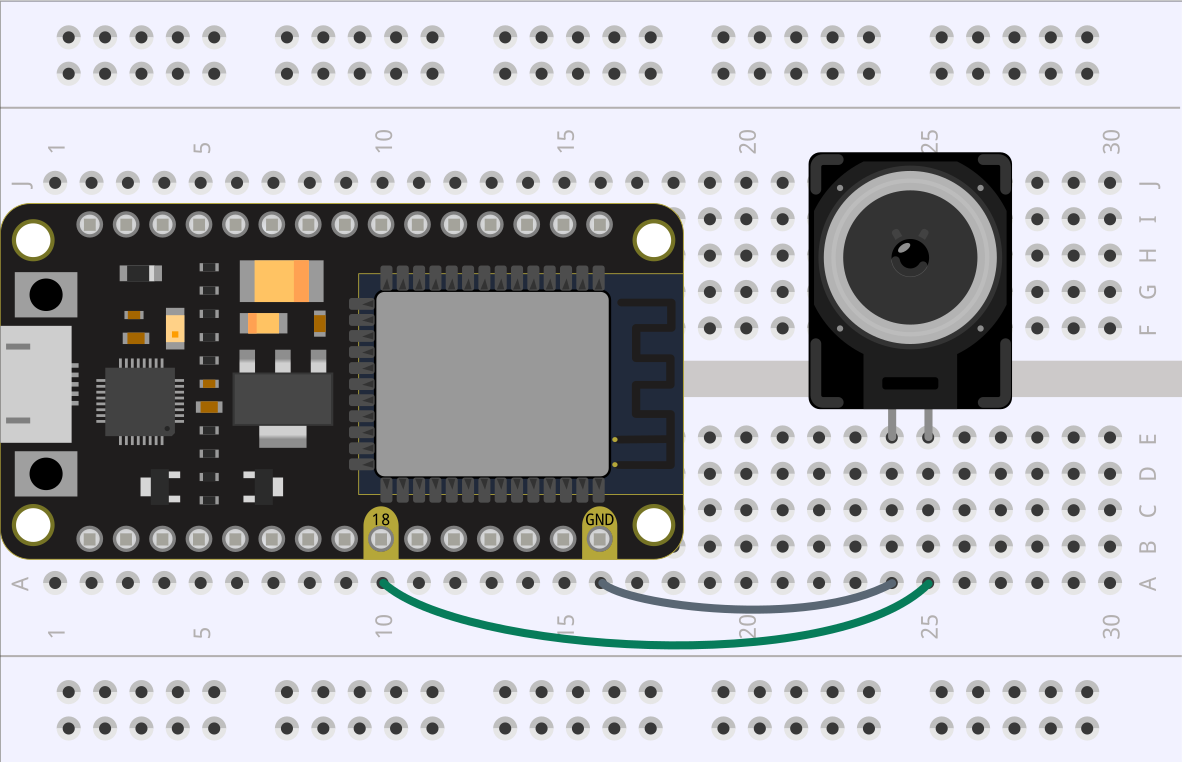
Un último apunte sobre el material. Seguro que en algún rincón tenemos un teléfono o un despertador que no funcionan, o quizás algún juguete viejo de esos que emiten una musiquilla electrónica. Aunque los piezos y los altavoces son muy baratos y fáciles de encontrar, en lugar de comprarlos podemos recuperarlos de estos dispositivos de desecho. Además, la calidad del sonido de un altavoz reciclado va a ser probablemente mucho mejor que la que se obtenga con cualquier piezo nuevo. De esta manera se le da una segunda vida a un aparato que acabaría convertido en basura electrónica.



Altavoz reciclado de un juguete viejo, con cables de conexión soldados.

8.3 El circuito

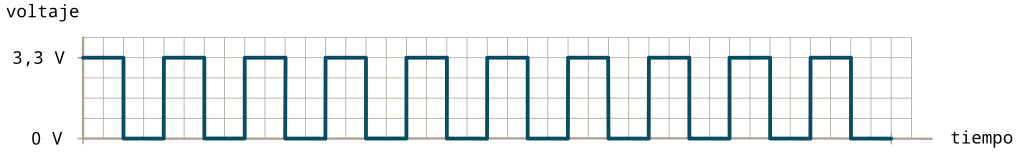
Los sonidos que vamos a crear con el microcontrolador los oiremos a través de un altavoz o un piezo. Estos dispositivos solo tienen dos terminales. Las conexiones, por tanto, son muy sencillas: un terminal se conecta a un GPIO de la placa y el otro a GND. En los ejemplos usaremos el GPIO18, aunque se puede escoger otro.



8.4 440 Hz

Una vez conectado el altavoz queremos que suene. Dado que el sonido es una onda, para generar un sonido con el microcontrolador debemos crear una onda. Y como una onda no es otra cosa que una señal periódica, eso es algo que sabemos conseguir usando la técnica de modulación por ancho de pulso o PWM que vimos en el capítulo anterior.

Pongámonos manos a la obra. Vamos a generar una nota de 440 Hz –hemos escogido este valor porque es el estándar de referencia para afinar la altura musical–. Para lograrlo simplemente tenemos que generar una señal PWM de esa frecuencia, para lo que recurriremos a la clase PWM del módulo machine.



Una señal periódica generada en el microcontrolador con PWM. El hecho de que la señal sea cuadrada produce el timbre característico de los sonidos electrónicos.

Las instrucciones ya las conocemos del capítulo pasado. En el constructor especificaremos, además del pin de conexión, la frecuencia y el ciclo de trabajo de la señal. La frecuencia está clara: 440 Hz. ¿Y el ciclo de trabajo? Dado que queremos generar una señal cuadrada, el ciclo de trabajo debe ser del 50 % (dentro de un ciclo, la señal está la mitad del tiempo a 3,3 V y la otra mitad a 0 V); por tanto, haremos duty\_u16=32768 (alternativamente, en placas basadas en ESP32 también se puede hacer duty=512 o, incluso, podemos no indicar este valor en el constructor ya que, por defecto, la señal se genera con un ciclo de trabajo del 50%).

>>> from machine import Pin, PWM

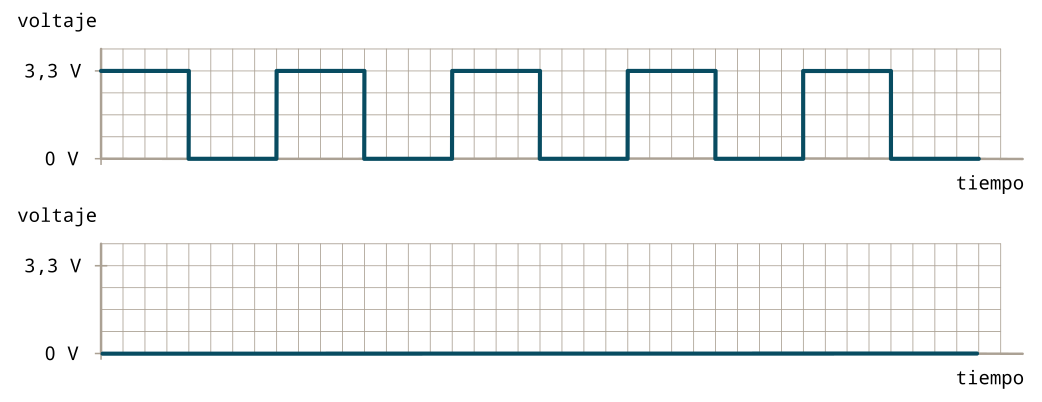
>>> ALTAVOZ = PWM(Pin(18), freq=440, duty\_u16=32768)

Si el altavoz está conectado al GPIO18, se emite la nota sin parar. Como vimos, una vez inicializado el pin PWM el hardware se encarga de mantener la señal (se puede parar pulsando Stop en el editor). No puede ser más sencillo crear notas musicales con el microcontrolador.

8.5 El ciclo de trabajo

Antes de seguir debemos preguntarnos cómo afecta el ciclo de trabajo de la señal a la nota musical. El ciclo de trabajo que utilizamos en el ejemplo anterior era del 50 %. Vamos a probar ahora con otros valores.

Si el ciclo de trabajo es 0, la señal PWM está todo el rato en estado apagado y, como cabría esperar, no se oye ningún sonido. De hecho, esta es la manera en la que silenciaremos las notas musicales en una composición.



Arriba: Una señal con un ciclo de trabajo del 50 %. Abajo: Un ciclo de trabajo del 0 % hace que la señal esté siempre a 0 V, por lo que no se genera sonido.

Por otro lado, el valor máximo del ciclo de trabajo hace que la señal esté continuamente en estado encendido; en consecuencia tampoco se genera sonido. La razón ahora es que, al estar siempre en 3,3 V, no se produce ninguna vibración en la membrana del altavoz, y sin vibración no hay sonido.

¿Y qué sucede con otros valores intermedios? Probemos, por ejemplo, con un valor del ciclo de trabajo del 25 % (duty\_16=16384). Si no se cambia la frecuencia, la nota sigue siendo la misma, aunque suena un poco diferente. Ha cambiado ligeramente el timbre del sonido porque al cambiar el ciclo de trabajo se ha variado la forma de la onda.

En conclusión, en el resto del capítulo los dos únicos valores que usaremos del ciclo de trabajo son 50 % (duty\_u16=32768 o duty=512) para crear un sonido y cero (duty\_u16=0 o duty=0) para silenciarlo.

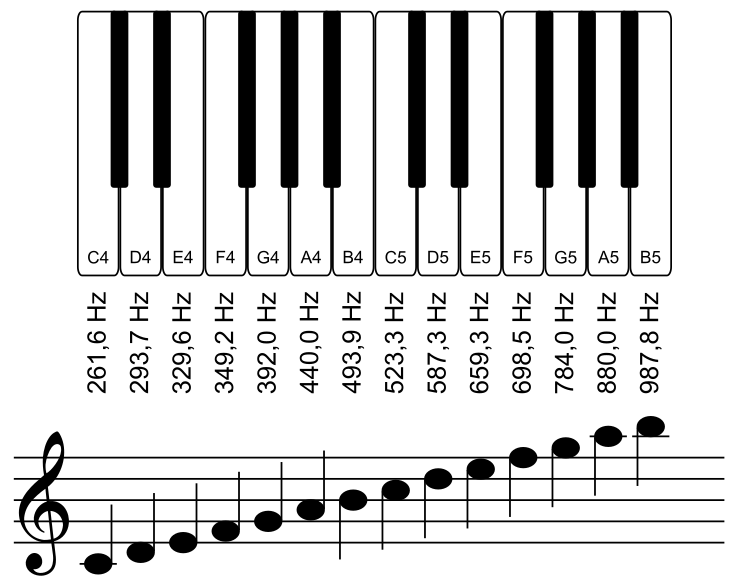
>>> ALTAVOZ.duty\_u16(32768)

>>> ALTAVOZ.duty\_u16(0)

8.6 Do, re, mi

En los apartados anteriores hemos sentado las bases de la generación de sonido con MicroPython: para generar un sonido crearemos una señal PWM cuya frecuencia determina el tono de la nota musical; el ciclo de trabajo, a su vez, lo pondremos al 50 %, excepto en los momentos en los que queramos silenciar el sonido, en cuyo caso lo pondremos al 0 %. Y ahora vamos a aplicarlo para obtener nuestras primeras melodías electrónicas.

Dado que ya sabemos cómo reproducir una nota musical, planteamos aquí el primer reto: escribir un programa para que el altavoz emita varias notas seguidas. Por ejemplo, do, re, mi, cuyas frecuencias respectivas son 262, 294 y 330 hercios (hemos redondeado los valores porque la frecuencia debe ser un número entero). La duración de cada nota es indiferente; ya nos ocuparemos más adelante de ajustarla. Como siempre, se debe intentar antes de consultar la respuesta.



Frecuencia de algunas notas musicales.

Una posible solución es la que mostramos a continuación.

from machine import Pin, PWM

from time import sleep\_ms

ALTAVOZ = PWM(Pin(18), duty\_u16=32768)

DO = 262

RE = 294

MI = 330

DURACION = 500

ALTAVOZ.freq(DO)

sleep\_ms(DURACION)

ALTAVOZ.freq(RE)

sleep\_ms(DURACION)

ALTAVOZ.freq(MI)

sleep\_ms(DURACION)

ALTAVOZ.deinit()

En el constructor hemos inicializado la señal PWM con un ciclo de trabajo del 50 % pero hemos dejado sin establecer el valor de la frecuencia, que asignamos posteriormente con el método freq(). Para la frecuencia de cada una de las notas hemos creado sendas variables, DO, RE y MI, y la duración de las notas la establecemos en una variable llamada DURACION. Finalmente desactivamos la señal para que se deje de emitir sonido.

8.7 ¿*Legato* o *staccato*?

En el programa anterior las notas se interpretaban unidas, una inmediatamente a continuación de la otra; esta manera de interpretar un grupo de notas musicales se llama *legato*. Existe otra técnica que consiste en interpretar las notas separadas por un breve silencio; es el llamado *staccato*. Para conseguirlo, se acorta la nota respecto a su valor original y así queda separada por un silencio de la nota siguiente en la melodía. Eso es lo que vamos a hacer ahora: interpretar las notas do, re, mi con un silencio entre ellas.

Aunque hay muchas maneras de implementar el programa, lo más conveniente es usar funciones. En un pentagrama, cada nota queda determinada por su altura (la frecuencia) y su duración; así mismo, los silencios se caracterizan por su duración. Por ese motivo crearemos dos funciones: una para interpretar una nota, con dos argumentos –la frecuencia y la duración de la nota–, y otra para el silencio, con un único argumento –la duración del silencio–. De esta manera, más adelante podremos aprovechar estas funciones para escribir composiciones musicales más elaboradas.

from machine import Pin, PWM

from time import sleep\_ms

ALTAVOZ = PWM(Pin(18), duty\_u16=0)

# Frecuencia de las notas musicales (Hz)

DO = 262

RE = 294

MI = 330

# Duración de las notas y del silencio (ms)

DURACION\_NOTA = 250

DURACION\_SILENCIO = 250

# Nota musical

def nota(frecuencia, duracion):

ALTAVOZ.duty\_u16(32768)

ALTAVOZ.freq(frecuencia)

sleep\_ms(duracion)

# Silencio

def silencio(duracion):

ALTAVOZ.duty\_u16(0)

sleep\_ms(duracion)

# Se desactiva la señal

def apagado():

ALTAVOZ.deinit()

# Melodía

nota(DO, DURACION\_NOTA)

silencio(DURACION\_SILENCIO)

nota(RE, DURACION\_NOTA)

silencio(DURACION\_SILENCIO)

nota(MI, DURACION\_NOTA)

silencio(DURACION\_SILENCIO)

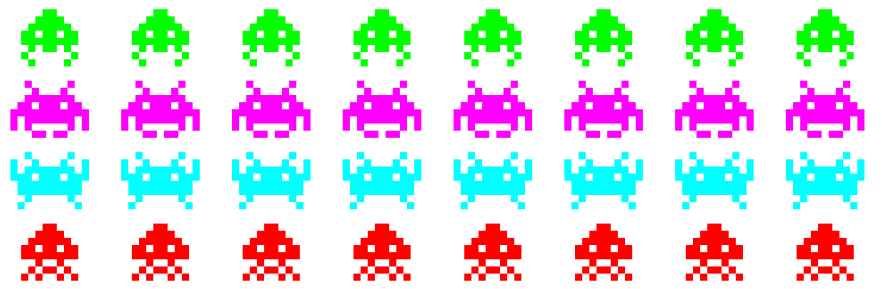
# Se desactiva el altavoz

apagado()

Para hacer este programa no es necesario utilizar funciones –se podrían haber escrito las instrucciones una a continuación de la otra–, pero es una buena práctica hacerlo así.

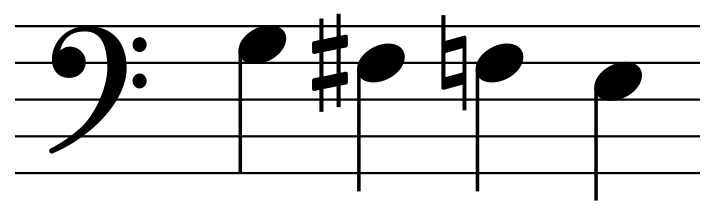
8.8 Space Invaders

En el año 1978 sale al mercado uno de los videojuegos más icónicos de todos los tiempos: Space Invaders. Este es el primer videojuego en el que la música interactúa con el desarrollo del juego. Aunque son solo cuatro notas que se repiten una y otra vez, no se trata de meros efectos sonoros; a medida que los enemigos se van moviendo a más velocidad, la melodía también acelera. La música, machacona y amenazadora, genera así una urgencia que produce desasosiego y angustia en el jugador.



Space Invaders, de la compañía japonesa Taito, fue el primer videojuego en el que la música interactuaba con la acción del juego.

Vamos a programar la secuencia de cuatro notas sol, fa#, fa, mi del juego Space Invaders. Las frecuencias de las notas son 165 Hz (mi), 175 Hz (fa), 185 Hz (fa sostenido) y 196 Hz (sol).



Las cuatro notas sol, fa#, fa, mi de Space Invaders.

Reutilizaremos las funciones del apartado anterior, y escogeremos la duración de las notas y los silencios de manera que “suene bien”. Para que la melodía se repita en bucle la incluiremos dentro de un bucle while True.

from machine import Pin, PWM

from time import sleep\_ms

ALTAVOZ = PWM(Pin(18), duty\_u16=0)

# Frecuencia de las notas musicales (Hz)

MI = 165

FA = 175

FA\_S = 185

SOL = 196

# Duración de las notas y del silencio (ms)

DURACION\_NOTA = 400

DURACION\_SILENCIO = 200

# Nota musical

def nota(frecuencia, duracion):

ALTAVOZ.duty\_u16(32768)

ALTAVOZ.freq(frecuencia)

sleep\_ms(duracion)

# Silencio

def silencio(duracion):

ALTAVOZ.duty\_u16(0)

sleep\_ms(duracion)

# Melodía

while True:

nota(SOL, DURACION\_NOTA)

silencio(DURACION\_SILENCIO)

nota(FA\_S, DURACION\_NOTA)

silencio(DURACION\_SILENCIO)

nota(FA, DURACION\_NOTA)

silencio(DURACION\_SILENCIO)

nota(MI, DURACION\_NOTA)

silencio(DURACION\_SILENCIO)

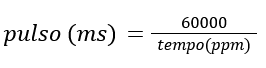
8.9 El *tempo* de la composición

Por ahora todo va muy bien. Pero en Space Invaders, según va avanzando el juego los aliens están cada vez más cerca y la melodía se reproduce más rápidamente. Si queremos modificar el programa para reflejar esta situación habría que cambiar las duraciones tanto de las notas como de los silencios, pero hacer esto a ojo es complicado y poco efectivo. A cambio, existe un término musical que nos va a ayudar a acelerar la música fácilmente: el *tempo*.

*Adagio*, *andante*, *allegro*, *presto*… Estas descripciones en una composición musical indican el *tempo*, es decir, la velocidad con la que debe ser interpretada la pieza. El *tempo* se expresa dando el número de pulsos (o *beats*) que se deben ejecutar por minuto. La unidad para medir el tempo es pulsos por minuto, que abreviaremos como ppm (o bpm, del inglés *beats per minute*). Sin profundizar demasiado podemos decir que un pulso es la duración de una figura determinada –normalmente la negra– que se toma como referencia. En consecuencia, un tempo de 60 pulsos por minuto indica que, en cada minuto de interpretación, hay que interpretar exactamente 60 pulsos. Esto equivale a decir que cada pulso debe tener una duración de un segundo. Lógicamente, cuanto mayor sea el número de pulsos por minuto más rápidamente debe interpretarse la pieza. O, dicho de otra manera, cuantos más pulsos por minuto, menos tiempo dura cada pulso.

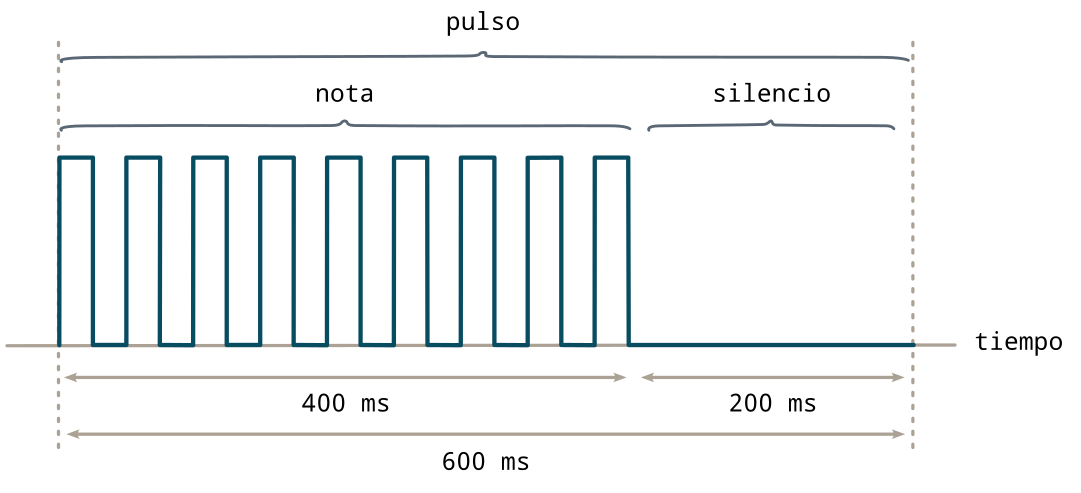
Veamos cómo podemos reescribir el programa anterior usando el *tempo* de la composición. La melodía de Space Invaders empieza relativamente lenta; podríamos caracterizarla como un *andante* o un *moderato*. Lo primero que necesitamos es transformar estas descripciones subjetivas en un número, y para ello se puede utilizar un metrónomo *online*. Un tempo de 100 ppm parece una buena elección para el comienzo de la melodía.

Necesitamos ahora saber cuánto tiempo dura un pulso. Existe una sencilla relación matemática entre el *tempo* y la duración de un pulso: dado que el *tempo* en pulsos por minuto es el número de pulsos que hay en un minuto, y un minuto tiene 60 segundos, un pulso dura 60/*tempo*, en segundos. O, multiplicando por 1000, la duración de un pulso es 60000/*tempo*, en milisegundos:



El tempo que hemos elegido es de 100 ppm, o 100 pulsos por minuto, por tanto la duración de un pulso, en milisegundos, será 60000/100 = 600 ms.

Ya sabemos, entonces, que para nuestra melodía a 100 ppm cada pulso debe durar 600 ms. Ahora bien, ¿en qué consiste exactamente un pulso? En nuestra composición podemos considerar que cada pulso está compuesto por la nota más el silencio. Por tanto, tenemos que repartir los 600 ms entre la duración de la nota y la duración del silencio. Dado que en el ejemplo habíamos puesto 400 ms como duración de la nota y 200 ms como duración del silencio, vamos a seguir con esos valores. Para hacerlo de manera sistemática, podemos decir que la nota dura las dos terceras partes del pulso y el silencio dura el tercio restante.



En un pulso de 600 ms, la nota ocupa los dos tercios del tiempo (400 ms) y el silencio un tercio (200 ms).

Ya estamos en condiciones de reescribir el programa usando el tempo de la composición en lugar de las duraciones de la nota y el silencio. Pero antes hagamos una pequeña comprobación, para asegurarnos de que estamos haciendo las cosas bien: vamos a escribir en el REPL las instrucciones para calcular los tiempos. Establecemos la variable TEMPO con el valor del tempo en pulsos por minuto (ppm).

>>> TEMPO = 100

Ahora calculamos la duración del pulso, la nota y el silencio:

>>> pulso = 60000/TEMPO

>>> duracion\_nota = 2/3\*pulso

>>> duracion\_silencio = 1/3\*pulso

e imprimimos sus valores:

>>> pulso

600.0

>>> duracion\_nota

400.0

>>> duracion\_silencio

200.0

Las duraciones son las deseadas, pero hay un problema: es muy probable que a la función sleep\_ms() no le guste que se le pasen estos valores como argumento. De hecho, si lo intentamos recibimos un mensaje de error.

>>> from time import sleep\_ms

>>> sleep\_ms(200.0)

TypeError: can't convert float to int

El problema es que los resultados obtenidos son del tipo float (es decir, son números reales) mientras que el argumento de sleep\_ms() debe ser un número entero. Por tanto, debemos recurrir a la función int() para transformar los float en int.

>>> duracion\_nota = int(2/3\*pulso)

>>> duracion\_silencio = int(1/3\*pulso)

Los valores son ahora números enteros, como necesitamos.

>>> duracion\_nota

400

>>> duracion\_silencio

200

Ahora ya podemos pasar a modificar el programa de la música de Space Invaders para incluir una variable con el *tempo* de la composición; también es conveniente crear una nueva función para generar la melodía. La modificación queda como ejercicio, ya que es inmediata. Una vez que esté listo, cambiando únicamente el valor del tempo se obtiene una melodía más lenta o más rápida. Probemos con un tempo de 200 ppm… ¡los enemigos están muy cerca!

8.10 ¡Ya están aquí!

El hecho de introducir el tempo en la composición nos permite hacer otra mejora en la música de Space Invaders. Según va avanzando el juego, la música se interpreta cada vez más y más rápido. Es lo que, en términos musicales, se llama *accelerando*. Modifiquemos el programa para que la música empiece lentamente –por ejemplo, a 80 ppm– y vaya acelerando poco a poco hasta llegar a un *prestissimo* –300 ppm–. Adjuntamos una solución pero, como siempre, es mejor intentarlo antes de mirarla.

from machine import Pin, PWM

from time import sleep\_ms

ALTAVOZ = PWM(Pin(18), duty\_u16=0)

# Frecuencia de las notas musicales (Hz)

MI = 165

FA = 175

FA\_S = 185

SOL = 196

# Tempo inicial (ppm)

TEMPO\_0 = 80

# Tempo máximo (ppm)

TEMPO\_MAX = 300

# Incremento en el tempo (ppm)

INCREMENTO = 10

# Se inicializa la variable tempo (bpm)

tempo = TEMPO\_0

# Nota musical

def nota(frecuencia, duracion):

ALTAVOZ.duty\_u16(32768)

ALTAVOZ.freq(frecuencia)

sleep\_ms(duracion)

# Silencio

def silencio(duracion):

ALTAVOZ.duty\_u16(0)

sleep\_ms(duracion)

# Melodía en función del tempo t

def melodia(t):

pulso = 60000/t

duracion\_nota = int(2/3\*pulso)

duracion\_silencio = int(1/3\*pulso)

# Se reproducen las notas

nota(SOL, duracion\_nota)

silencio(duracion\_silencio)

nota(FA\_S, duracion\_nota)

silencio(duracion\_silencio)

nota(FA, duracion\_nota)

silencio(duracion\_silencio)

nota(MI, duracion\_nota)

silencio(duracion\_silencio)

while True:

while tempo <= TEMPO\_MAX:

# Se interpreta la melodía

melodia(tempo)

# El tempo se incrementa

tempo += INCREMENTO

# Se vuelve al tempo inicial

tempo = TEMPO\_0

# Se espera 1 s antes de volver a empezar

sleep\_ms(1000)

8.11 *Korobéiniki*

En 1989 el compositor japonés Hirokazu Tanaka creó el archiconocido arreglo de la melodía popular rusa *Korobéiniki* que se incluyó como música “tipo A” en la versión de Tetris para la Game Boy de Nintendo. Vamos a reproducir con nuestro microcontrolador la que se convirtió en una de las melodías más famosas de la historia de los videojuegos. Pero para ello primero hay que conseguir la partitura.

Hay muchas adaptaciones en Internet de las melodías de los videojuegos más conocidos; se deben buscar los arreglos para un solo instrumento (no podemos reproducir melodías polifónicas) o, si tenemos conocimientos musicales, podemos hacer nuestros propios arreglos. Nosotros vamos a utilizar la versión para flauta disponible en la página www.flutetunes.com. En dicha página, además de la partitura de *Korobéiniki*, hay multitud de melodías en versiones muy adecuadas para la tarea que nos ocupa.

Con la partitura delante, tenemos que identificar las notas musicales para poder usarlas en el programa. Una nota musical queda caracterizada por la duración y por el tono (la frecuencia). La figura (redonda, blanca, negra…) indica la duración, y su situación en el pentagrama el tono (do, re, mi… y la octava). Con unos conocimientos musicales básicos –simplemente saber cómo leer una partitura– podemos identificar los tonos y las duraciones de cada nota. Si no sabemos, o si simplemente no nos apetece dedicar el tiempo a esa tarea, dejamos más abajo la transcripción de la partitura.



Dos compases de *Korobéiniki* en el arreglo para flauta de www.flutetunes.com. Debajo de cada figura se indica el tono de la nota y su duración.

El tono

El criterio que hemos seguido para nombrar cada nota es el siguiente: el nombre de la nota en minúsculas seguido, si es necesario, de la alteración que corresponda (B para bemol y S para sostenido) y finalmente un número que indica la octava. Por ejemplo, al do de la quinta octava le hemos llamado do5, al sol sostenido de la cuarta octava solS4, y así sucesivamente. A mayores, hemos usado el valor 0 para identificar los silencios. Siguiendo este criterio definimos las variables que usaremos en el programa con las frecuencias de las notas, en hercios.

solS4 = 415

la4 = 440

laS4 = 466

si4 = 494

do5 = 523

doS5 = 554

re5 = 587

reS5 = 622

mi5 = 659

fa5 = 698

faS5 = 740

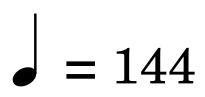
sol5 = 784

solS5 = 831

la5 = 880

La duración

La duración de la nota la especificamos con el nombre de la figura: blanca, con una duración de dos pulsos; negra, un pulso; corchea, medio pulso, y negra\_punt es la negra con puntillo, cuya duración es un pulso y medio. Hemos tomado como un pulso la duración de una negra, porque al principio de la partitura se indica que es así; se nos dice, además, que el tempo es de 144 pulsos por minuto.



La indicación del *tempo* al inicio de la partitura.

Estableciendo el tempo en 144 pulsos por minuto, a partir de este valor hallamos la duración de cada figura.

TEMPO = 144

pulso = 60000/TEMPO

blanca = int(2\*pulso)

negra = int(1\*pulso)

negra\_punt = int(1.5\*pulso)

corchea = int(0.5\*pulso)

La partitura

Lo que viene ahora es escribir la partitura usando las variables creadas. Para ello usaremos sendas listas, una para las frecuencias y otra para las duraciones de cada nota. La primera lista, que hemos llamado korobeiniki\_tono, contiene los tonos de las notas en el orden en que deben ser interpretados.

korobeiniki\_tono = [

mi5, si4, do5, re5, do5, si4, # Compás 1

la4, la4, do5, mi5, re5, do5, # Compás 2

si4, do5, re5, mi5, # Compás 3

do5, la4, la4, la4, si4, do5, # Compás 4

re5, fa5, la5, sol5, fa5, # Compás 5

mi5, do5, mi5, re5, do5, # Compás 6

si4, si4, do5, re5, mi5, # Compás 7

do5, la4, la4, 0, # Compás 8

mi5, si4, do5, re5, do5, si4, # Compás 1

la4, la4, do5, mi5, re5, do5, # Compás 2

si4, do5, re5, mi5, # Compás 3

do5, la4, la4, la4, si4, do5, # Compás 4

re5, fa5, la5, sol5, fa5, # Compás 5

mi5, do5, mi5, re5, do5, # Compás 6

si4, si4, do5, re5, mi5, # Compás 7

do5, la4, la4, 0, # Compás 8

mi5, do5, # Compás 9

re5, si4, # Compás 10

do5, la4, # Compás 11

solS4, si4, 0, # Compás 12

mi5, do5, # Compás 13

re5, si4, # Compás 14

do5, mi5, la5, # Compás 15

solS5, 0 # Compás 16

]

Análogamente, la lista korobeiniki\_duracion contiene la duración de cada una de las notas.

korobeiniki\_duracion = [

negra, corchea, corchea, negra, corchea, corchea,

negra, corchea, corchea, negra, corchea, corchea,

negra\_punt, corchea, negra, negra,

negra, negra, corchea, corchea, corchea, corchea,

negra\_punt, corchea, negra, corchea, corchea,

negra\_punt, corchea, negra, corchea, corchea,

negra, corchea, corchea, negra, negra,

negra, negra, negra, negra,

negra, corchea, corchea, negra, corchea, corchea,

negra, corchea, corchea, negra, corchea, corchea,

negra\_punt, corchea, negra, negra,

negra, negra, corchea, corchea, corchea, corchea,

negra\_punt, corchea, negra, corchea, corchea,

negra\_punt, corchea, negra, corchea, corchea,

negra, corchea, corchea, negra, negra,

negra, negra, negra, negra,

blanca, blanca,

blanca, blanca,

blanca, blanca,

blanca, negra, negra,

blanca, blanca,

blanca, blanca,

negra, negra, blanca,

blanca, blanca

]

Estas dos listas nos proporcionan la partitura de la melodía escrita de manera que la podamos usar en un programa. Antes de seguir adelante, es conveniente leer el siguiente apartado en el caso de que no conozcamos el uso de listas en Python.

8.12 Las listas en Python

Para escribir la partitura de *Korobéiniki* hemos guardado los valores del tono y la duración de cada nota musical en sendas listas. Probemos las siguientes instrucciones en el REPL para entender cómo vamos a proceder para reproducir la melodía.

Una lista en Python es simplemente una manera de agrupar varios datos bajo un mismo nombre. Los elementos de la lista se escriben entre corchetes y separados por comas. Tomemos un único compás para simplificar las cosas, por ejemplo el quinto. Por un lado, la lista con los tonos es:

>>> tono = ["Re", "Fa", "La", "Sol", "Fa"]

y la lista de las duraciones:

>>> duracion = ["Negra con puntillo", "Corchea", "Negra", "Corchea", "Corchea"]

Hemos puesto las comillas porque en este ejercicio solo nos interesa el nombre de los elementos, no su valor; es decir, los vamos a tratar como cadenas de caracteres y no como variables, de ahí las comillas.

Podemos imprimir el tono de primera nota de la lista, tono[0], y su duración, duracion[0] (en Python las listas empiezan en cero):

>>> tono[0]

'Re'

>>> duracion[0]

'Negra con puntillo'

La segunda nota tiene un tono tono[1] y dura duracion[1], y así sucesivamente. Es decir, para identificar cada nota tomamos un elemento de la lista tono y el elemento de la lista duracion que corresponde a su misma posición. Así de simple.



Para recorrer todos los elementos de una lista debemos saber primero cuántos elementos tiene, para lo que se usa la función len().

>>> len(tono)

5

Otra función fundamental cuando se trabaja con listas en Python es range(). Con un único argumento, range(N) –N tiene que ser entero– crea una secuencia de números enteros consecutivos entre 0 (incluido) y N (excluido). Por tanto, range(len(tono)) crea la secuencia de números 0, 1, 2, 3 y 4.

Para iterar por todos los valores de una lista se usa normalmente el bucle for. Así, para obtener todas las notas de nuestro compás podemos usar las siguientes instrucciones:

>>> for i in range(len(tono)):

... print(i, "\t", tono[i], "\t", duracion[i])

0 Re Negra con puntillo

1 Fa Corchea

2 La Negra

3 Sol Corchea

4 Fa Corchea

Una vez vista la manera de trabajar con listas, estamos ya en condiciones de escribir la melodía.

8.13 La melodía del Tetris

Ya tenemos la partitura de *Korobéiniki* escrita en “versión Python”. Para crear la melodía vamos a reutilizar las funciones de los apartados anteriores que nos permitían crear una nota (dada la frecuencia y el tiempo de duración) y un silencio (dada la duración).

def nota(frecuencia, tiempo):

ALTAVOZ.duty\_u16(32768)

ALTAVOZ.freq(frecuencia)

sleep\_ms(tiempo)

def silencio(tiempo):

ALTAVOZ.duty\_u16(0)

sleep\_ms(tiempo)

Usando estas funciones, para escribir la melodía no tenemos más que ir “interpretando” las notas una a una. Para ello llamamos a la función nota() con los argumentos de frecuencia y tiempo dados por cada par de valores korobeiniki\_tono[i] y korobeiniki\_duracion[i] de las listas definidas anteriormente. Hay que añadir también un pequeño silencio entre nota y nota; de lo contrario, dos notas iguales consecutivas sonarían como una sola.

Sin embargo, antes hay que hacer una comprobación, ya que también hay que interpretar silencios; en tal caso –cuando en el tono tiene el valor 0–, en lugar de llamar a la función nota() llamaremos a silencio().

for i in range(len(korobeiniki\_tono)):

if korobeiniki\_tono[i] == 0:

silencio(korobeiniki\_duracion[i])

else:

nota(korobeiniki\_tono[i], korobeiniki\_duracion[i])

silencio(10)

Solo resta poner juntas todas las instrucciones que hemos ido viendo en este apartado y en los anteriores; a saber:

* Importación de los módulos necesarios.
* Inicialización del altavoz.
* Listado de las variables (frecuencias y duraciones).
* Listas con los tonos y las duraciones.
* Definición de las funciones nota() y silencio().
* Bucle for en el que se interpreta la melodía.

Una vez que hemos juntado todas las piezas en un único programa escuchamos el resultado. No está nada mal, ¿verdad?

Con este ejercicio se han sentado las bases para interpretar cualquier otra melodía. Solo hay que “traducir” la partitura a tonos y duraciones y ajustar el tempo; el programa es totalmente reutilizable.

8.14 Archivos externos

En este último apartado no introduciremos nada nuevo relacionado con la creación musical, sino que trataremos algo que nos puede ayudar en cualquier programa de MicroPython.

Gracias a lo aprendido en los apartados anteriores podemos reproducir con nuestro microcontrolador la melodía que deseemos. Las partituras de las distintas melodías son, obviamente, diferentes. Por tanto, las notas utilizadas también cambiarán. En el caso de *Korobéiniki* hemos incluido los valores de las frecuencias en el programa porque el rango de notas era pequeño (estaban todas comprendidas entre el sol sostenido de la cuarta octava y el la de la quinta). Pero buscar cuáles son las notas que aparecen en cada melodía para incluirlas en el programa es poco práctico. Podíamos también proceder de la siguiente manera: copiamos en el programa la lista de las frecuencias de todas las notas, estén o no en la partitura, y ya se usarán las que hagan falta. No es descabellado, pero quedaría un programa demasiado largo. Hay una solución parecida a esta pero que hace que el tamaño programa, lejos de aumentar, disminuya: incluir todas las frecuencias –sean o no necesarias para la melodía en cuestión– en un archivo externo.

Veamos cómo lo podemos hacer. En el editor creamos un nuevo archivo, con el nombre frecuencias.py, y en él escribimos las variables que representan cada tono junto al valor de su frecuencia (estos valores debemos buscarlos en alguna referencia *online*). Mostramos aquí algunas de las líneas del archivo.

...

mi4 = 330

fa4 = 349

faS4 = 370

solB4 = 370

sol4 = 392

solS4 = 415

laB4 = 415

la4 = 440

...

Este archivo, una vez creado, lo guardamos en el sistema de archivos de la placa (el procedimiento es el mismo que usamos en capítulos anteriores). Una vez que se ha guardado el archivo en la memoria *flash* del microcontrolador podemos acceder a él desde el programa igual que si fuese uno de los módulos integrados de MicroPython: usando la instrucción import frecuencias (sin la extensión). Vamos a hacer primero un par de pruebas desde el REPL.

>>> import frecuencias

Si no obtenemos un mensaje de error, es que el archivo se importó correctamente. Veamos ahora si podemos acceder a las variables que contiene el archivo; por ejemplo a la que tiene el nombre la4, cuyo valor es 440.

>>> la4

NameError: name 'la4' isn't defined

Este mensaje de error no debería sorprendernos. Cuando importábamos las funciones de un módulo anteponíamos a la función el nombre del módulo, separado por un punto. Probémoslo.

>>> frecuencias.la4

440

Ahora funciona a la perfección. Claro que no parece muy cómodo escribir todas las notas musicales de esta manera. Probemos otra manera de hacerlo: importando solo la nota.

>>> from frecuencias import la4

>>> la4

440

Haciéndolo así llega con poner solo el nombre de la variable, pero tampoco parece muy buen negocio: tenemos que importar todas las variables, una a una, y eso es precisamente lo que intentábamos evitar. Sin embargo, hay una manera de no tener que escribir los nombres de todas las variables, y es usando un asterisco, \*. Este asterisco es equivalente a decir “todas las variables, funciones, etc. que existan dentro del módulo en cuestión”. Y eso es lo que vamos a hacer:

>>> from frecuencias import \*

>>> la4

440

¡Perfecto! Podemos acceder a las variables del archivo frecuencias.py sin necesidad de importar las variables una a una, y sin tener que escribir el nombre del archivo delante de cada variable. En el programa donde se reproducía *Korobéiniki* añadimos la instrucción from frecuencias import \* y borramos las declaraciones de las variables correspondientes a los tonos; el programa debería funcionar exactamente igual.

Aunque en este caso es una buena solución, cuidado con el asterisco. La importación de las funciones y variables es completamente a ciegas; si se da la circunstancia que dos módulos tengan una función con el mismo nombre pero que realiza tareas diferentes, al utilizarla no sabemos cuál de ellas estamos usando, lo cual puede ser un gran problema.

Esta estrategia de separar el código en diferentes archivos es muy útil para mantener los programas organizados y con tamaños razonables. Además permite reutilizar código: las variables del archivo frecuencias.py están disponibles para ser utilizadas desde cualquier programa.

En resumen

Dado que el hilo conductor de este capítulo ha sido el sonido, hemos empezado relacionando las ondas sonoras con las señales PWM que introdujimos en el capítulo anterior. Hemos aprendido cómo crear notas musicales de diferente tono y duración, y hemos unido varias notas para interpretar una composición musical. Hemos visto cómo se puede establecer el *tempo* de una composición musical y lo hemos modificado para generar una melodía que se interpreta cada vez más rápido. Para finalizar, hemos dividido el código en distintos archivos para crear un programa más pequeño y organizado. Con todo lo aprendido podremos reproducir la música de cualquier videojuego de arcade y otras muchas melodías.